Push Job Specification

Christopher Brown <cb@opscode.com>
Kevin Smith <kevin@opscode.com>

May 1, 2012

1 Overview

This specification describes the ability to execute on-demand chef-client via knife. For brevity's sake this feature will be referred to as "push job" or "push jobs".
The concept of push jobs is quite simple. A user is able to select some subset of nodes managed by their Private Chef organization, specify an action or command for those nodes to execute, and track the status of each node as it executes the request.
A bit of complexity and bookkeeping lurks underneath push job's simplicity. Managed nodes will need to a way to reliably monitor the job coordinators so they can cope with outages. Job coordinators also need to monitor the status of each managed node so commands are only sent to available nodes and to track job progress.
The push job server and the managed nodes keep tabs on each other via a bidirectional heartbeating system. The push job server sends heartbeat messages to all the managed nodes, and the managed nodes send heartbeat messages to the server.
The remainder of this document will attempt to describe this feature in enough detail to allow a reasonable and scalable implementation.

2 Assumptions

2.1 Connectivity

1. Managed nodes **MUST** be reachable via a TCP-enabled network interface. 2. Managed nodes **MUST** be able to accept incoming TCP connections. 3. Managed nodes **MUST** be able to connect to the heartbeat and job coordination components inside Chef server.

2.2 Data format & Storage

1. All messages will be formatted as legal JSON. 2. The database is the canonical store of all application data.

2.3 Scalability & Security

1. Push jobs will be deployed in Private Chef only. 2. Push jobs will not be deployed in Hosted Chef. 3. The design must scale up to 8,000 managed nodes per OPC server. 4. Push jobs will honor the same security guarantees made by the Chef REST API.

3 Architecture

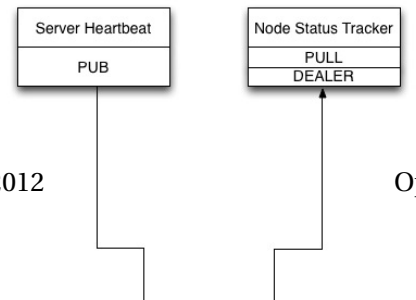
3.1 Communications

Managed nodes and server components will communicate using ZeroMQ messaging. While the server will be bound to predefined ports, the client should never require them. This is to ease running multiple clients on the same host for scalability testing; we will want to be able to run many hundreds of clients on the same machine rather than stand up thousands of individual nodes to test message scalability.
Communication can be separated into two categories: heartbeat and job execution. The heartbeat channel is used by the Chef server to detect when managed nodes are offline and, therefore, unavailable for job execution. Managed nodes also use the heartbeat channel to detect when the server is unavailable.
The job execution channel is used by the Chef server to send job execution requests to managed nodes. Managed nodes used the execution channel to send job-related messages such as acknowledging jobs, sending progress updates, and reporting final results.

3.1.1 Heartbeat Channel

PUB and SUB sockets are used because they automatically and scalably manage the fanin the server requires to monitor nodes as well as the fanout the server needs to broadcast its heartbeat to all the nodes.
In the interest of simplicity the heartbeat protocol should be as stateless as possible. The server continually broadcasts heartbeat messages to all clients that are listening. Clients indicate their availability by sending heartbeat messages to the server. There is no process beyond the zeromq connection to start or stop a connection from the client.
This might be worth modifying in the future. For example we might want to lower the signature validation load on the server by having a per-session symmetric key for the client heartbeat established at startup. Instead of the somewhat expensive public key signature check we could simply decrypt the packet with the session key and check for sanity. (TODO: think about whether this protocol is actually sane and secure)

The details of how nodes and servers discover and connect to each other's PUB and SUB sockets is covered in Server and Client Discovery.



3.1.2 Command Channel

TBD

3.2 Configuration/Discovery process

The configuration and service discovery process will provide the following pieces of data:

- The push job server hostname or address
- The port to subscribe to for server heartbeat
- The port to push client heartbeats to
- The public key of the server
- The lifetime of this configuration information

A configuration endpoint ':organization/nodes/push_jobs' will be added to our chef rest services. A signed GET to this endpoint will retrieve the appropriate configuration information in JSON format.

```
{
  "type": "co
  "host": "op
  "push_jobs"

  "public_key
```

```
                                                                    "lifetime"

                                                                }
```

type message type
host sender's host name (Private Chef server)
push_jobs/heartbeat/out_addr URL pointing to the server's heartbeat broadcast service
push_jobs/heartbeat/in_addr URL pointing to the server's node state tracking service
push_jobs/interval Interval, in seconds, between heartbeat messages
push_jobs/offline_threshold How many intervals must be missed before the other end is considered offline
push_jobs/online_threshold How many intervals must be missed before the other end is considered online
public_key The signing key that the push server will use.
lifetime how long in seconds this configuration is good for. The client should reload the configuration information after this has expired.

We may wish to use the discovery process to handle failover to a new server and distribution of clients among multiple servers. The discovery system would allocate the clients to various active servers and if a client lost the server heartbeat for a certain length of time (or got a reconfigure command via the command channel) it would reload the configuration and start a connection to the appropriate server. We would also reconfigure after the liftime of the configuration expires.

3.3 General Messaging

3.3.1 JSON

Push jobs use JSON because ZeroMQ handles packet fragmentation and reassembly. JSON also facilitates easier debugging and maintenance of the system since all messages are textual and human-readable. A binary protocol, such as Protocol Buffers or msgpack, would be more efficient but would also substantially increase the effort required to debug and support the system.

3.3.2 Security

All messages are signed using the caller's private key. This signature is transmitted in a separate ZeroMQ frame before the JSON payload.[1]

```
Sock = connect_to_server("tcp://some_server:8765"),
Sig = sign_message(JSON),
erlzmq:send(Sock, Sig, [sndmore]),
erlzmq:send(Sock, JSON)
```

[1] Public key signatures are used to verify the sender's identity and provide some amount of message tamper detection.

3.3.3 Heartbeat

```
{
  "type":  "heartbeat",
  "host":  "node123.foo.com",
}
```

type message type
host the sender's hostname

3.4 Protocols

3.4.1 Heartbeat

Liveness detection in a distributed system is a notoriously difficult problem. The most common approach is to arrange for two parties to exchange heartbeat messages on a regular interval. Let's call these two parties 'A' and 'B'. Both A and B are considered 'online' while they are able to exchange heartbeat messages. If A fails to receive heartbeats from B for some number of consecutive intervals then A will consider B 'offline' and not route any traffic to B. A will update B's status to 'online' once A starts receiving heartbeats from B again.
The protocol described here is loosely based on the Paranoid Pirate Protocol, but with some embelishments introduced because of the need for signing.
The heartbeat server sends out regular heartbeats to managed nodes via ZeroMQ PubSub. Managed nodes send their heartbeats over a separate channel. See the Heartbeat Channel section for a visual representation of the message flows and ZeroMQ sockets.

3.4.2 Message Format

The basic message format used here is a simple header packet containing the protocol version and a signature separated by CRLF.
The main packet is a JSON blob. Push jobs use JSON because ZeroMQ handles packet fragmentation and reassembly. JSON also facilitates easier debugging and maintenance of the system since all messages are textual and human-readable. A binary protocol, such as Protocol Buffers or msgpack, would be more efficient but would also substantially increase the effort required to debug and support the system.

3.4.3 Security

All messages are signed using the caller's private key. This signature is transmitted in a separate ZeroMQ frame before the JSON payload. The actual payload is not encrypted, and is broad cast to all clients. The system should never broadcast any data that is sensitive, such as commands or node status. This implies that the server heartbeat broadcast is not suitable for commands.

3.4.4 Socket configuration

The heartbeats (and other messages) flowing through the system are time sensitive. There is little value keeping many more packets than the online/offline threshold values. If we go too long without receiving a heartbeat, we will be declaring the machine down anyways. Furthermore, the signing protocol will mandate the rejection of aged packets.
This implies that the HWM values should be kept small, and ZMQ_SWAP should always be zero.

3.5 Server Heartbeat

3.6 Server Heartbeat Channel

PUB/SUB sockets are used for the server heartbeat because this manages the fanout required to send messages to thousands of clients. The client subscribes to the server heartbeat at a host/port combination specified in the discovery process.
The HWM should be kept small; there is no point in storing messages for dead clients any longer than necessary. Clients going down must be accepted and tolerated. If a client is not reachable for any length of time we want to drop those messages. This is in keeping with the fail fast philosphy.
The clients do not ACK the server heartbeats, and the server should not expect any.

3.6.1 Server Heartbeat Message

First packet (why not break each of these into packets and take advantage of 0mq's multi-part for parsing the header as well?)

```
VersionId\r\n            # a decimal ASCII integer value for the protocol version (1 for now)
SignedChecksum\r\n       # the signed checksum of the second packet in hexadecimal (base64 could be
```

Second packet is json

```
{"server":"SERVER",                # specifies the server
 "sequence":SEQUENCE_NUMBER",      # integer sequence number
 "timestamp":"TIMESTAMP",          # timestamp
 "type":"MSGTYPE"                  # 'heartbeat' for now
}
```

3.7 Client Heartbeat

3.8 Client Heartbeat Channel

PUSH/PULL sockets are used for the client hearbeat. The client PUSHes heartbeats to the server at the host/port specified in the config data received during discovery. The server will not ACK heartbeats.

NOTE: Some versions of this spec had PUB/SUB being used for this process. It would be simpler if the client was able to connect to the server to send heartbeats, rather than requiring the server to bind to the client. The latter would require some sort of handshake on startup to inform the server where to connect. While it is possible to bind the SUB to an address and connect the PUB, this seems to be not recommended (see zeromq guide, 'Getting the Message Out'). However, it seems that multiple PUSH to one PULL is supported, and we can bind the PULL socket to an address without trouble.
There isn't any reason we couldn't use the heartbeat to convey extra information. The public key signature based authentication process for heartbeats already requires a moderate sized payload, so a little extra information seems pretty harmless. This is in contrast to the 1-2 byte sized payload in the paranoid pirate protocol. Possible items to include are:

- The port the command processor is listening on.
- ID and status of the most recently received command.
- Information allowing the detection of crashed clients

3.8.1 Client Heartbeat Message

First packet

```
VersionId\r\n            # an decimal ascii integer value for the protocol version
SignedChecksum\r\n       # the signed checksum of the second packet in hexadecimal (base64 could be
```

Second packet is json

```
{"client":"CLIENTNAME",             # specifies which client key to use for signature check
 "org":"ORGNAME",                   # orgname of the client
 "sequence":SEQUENCE_NUMBER",       # integer sequence number
 "timestamp":"TIMESTAMP",           # timestamp
 "command_port":PORT                # the port we are listening on for commands
}
```

The client will discontinue the heartbeat and note the server as down if the server heartbeat state moves to down, and resume it when the server heartbeat resumes.
A managed node must mark the OPC server as offline when it fails to receive server heartbeats for a consecutive number of intervals equal to push_jobs/heartbeat/offline_threshold. A managed client must not attempt to send any data when the server is offline. Any job requests received by the managed node from the offline server must be discarded.
After a managed node has marked the server as offline it must receive server heartbeats for a consecutive number of intervals equal to push_jobs/heartbeat/online_threshold before marking the server online. The managed node may resume sending data and accepting job requests from the OPC server at this point.

If the client fails to recieve a heartbeat for too long, it will query the configuration interface to receive a possible configuration update. This would allow the system to recover from a failed server.
The client may wish to detect if the HWM is reached on the PUSH socket, since it will block when the HWM is reached. One strategy would be to set the HWM low and have some sort of alarm detect if we are blocked for any length of time. If the HWM is reached, we should declare the server down as if it stopped sending heartbeats.

3.9 Client-Server command channel

While it is outside the scope of this document, one viable approach for the command channel is for the client to bind a PULL socket to a port and pass that via the heartbeat to the server.

3.9.1 Jobs

A job is a request to execute a command sent to a group of managed nodes. The server is responsible for managing the execution of the job and its result set, if any. While job execution is conceptually simple it does require dealing with several aspects of distributed systems.

3.9.2 TODO Flesh out this section with CB's comments

3.9.3 TODO Describe a coherent design using ZeroMQ socket types

- Selecting servers
- Determining if enough servers are present to initiate the job
- Job initiation
- Tracking the progress of each participant
- Gathering participant job execution responses
- Returning job execution status & results to caller